IPFS as a foundation for anonymous file storage

Marc Guasch Repullés
Grau en Enginyeria Informàtica

Félix Freitag

01/2020

FITXA DEL TREBALL FINAL

Títol del treball:	IPFS as a foundation for anonymous file storage
Nom de l'autor:	Marc Guasch Repullés
Nom del consultor:	Félix Freitag
Data de lliurament (mm/aaaa):	*01/2020*
Àrea del Treball Final:	Sistemes distribuïts
Titulació:	*Grau en Enginyeria Informàtica*

Resum del Treball (màxim 250 paraules):

La intenció del treball és avaluar IPFS com a tecnologia, i situar-lo dins del context de l'estat de l'art quant a sistemes distribuïts. Un cop fet això, plantejar el disseny d'un servei d'emmagatzematge de fitxers, però que es recolzi en les capacitats de descentralització que ofereix IPFS, afegint capacitats d'anonimat per als usuaris i les seves dades.

Abstract (in English, 250 words or less):

The intention of the work is to evaluate IPFS as a technology, and place it within the context of the state of the art in terms of distributed systems. Once this is done, evaluate the design of a file storage service, but relying on the decentralization capabilities offered by IPFS, adding anonymity capabilities for users and their data.

Paraules clau (entre 4 i 8):

ipfs, descentralitzat, anonimitat, emmagatzematge de fitxers, decentralized, file storage, anonimity

Index

Figures index

1. Introduction

1.1. Context and justification

Nowadays the Internet has become part of everyone lives, in ways we do not even notice. We are constantly connected, identified and tracked by many companies and organizations for various reasons: advertisement, politics, data collection, … At the same time, this degree of insight third parties have over our data and online persona, is a risk for certain areas in the world, were governments can choose what data is allowed in their countries and even identify individuals that might be at risk for the information the consume or produce.

For this reason I believe the future of Internet is decentralized: move the data out of controlled siloes by companies and governments, and anonymous, so people can safely store and share files while keeping control of their data themselves.

Various efforts emerged during the years to improve security, anonymity, Internet neutrality and to combat censorship, but there are still areas worth to invest in. Any solution to become ubiquitous needs either to be easy to use by end users, or deeply rooted in the Internet protocols so end users do not have to care about them at all. So far there is no such solution to store your data in a long term, on the cloud way. To be able to store your personal files on the cloud, you need to grant third parties with full and centralized access to them, basically giving up control over it.

This work will try to design a proof of concept of a distributed file storage that can be easily embedded in end user applications, and that adds anonymity and private access to users data.

1.2. Goals

The goals of this work are to analyze IPFS, what are the fundamentals behind it, what are its strengths and weaknesses, and see if we can leverage it to build an anonymous file storage system on top of it that provides individuals with secure and anonymized storage to their files.

Once this evaluation is done, we will present a design proposal along with a proof of concept.

1.3. Approach and methodology

The project consists of two differentiated parts: first, a background gathering, which will try to get all the required context, both historical and technical, and from that, elaborate a design proposal for our system. The second part, will be organized around implementing a proof of concept of the designed solution.

The steps are:

1. Background research:
    a. Research about IPFS and its technical foundations
    b. IPFS internals and subsystems
    c. Give more context presenting other decentralized storages.

2. Requirement analysis:
    a. We scope the minimum requirements we want our system to fulfill
    b. We elaborate a requirements document and roadmap to achieve our goals

3. Design:
    a. We propose a design, providing required technical justification for it

4. Implementation:
    a. Implementation of the designed system in Go language
    b. Document subsystems as we elaborate them

5. Validation:
    a. Elaborate end to end tests that prove to validate the system is behaving as expected
    b. Document the results

6. Conclusions

1.4. Planification

To have a global roadmap of the project, we created a Gantt chart, were in a graphical way we summarize the high level deliverables and tasks. Those tasks are scoped in time and effort, to help us to organize better and achieve the proposed goals.

	TASK NAME	START DATE	DAY OF MONTH	END DATE	DURATION (WORK DAYS)
Contextualizing IPFS					
	IPFS history and background	4/10/19	4	7/10/19	3
	Analyize technology principles	8/10/19	8	14/10/19	6
	Compare to other technologies, strengths vs weaknesses	15/10/19	15	24/10/19	9
Design and proposal of application					
	Document about topic motivation	1/11/19	31	3/11/19	2
PAC2	Analyze technical challenges	4/11/19	34	8/11/19	4
	Technical framework b/w IPFS and proposal	11/11/19	41	18/11/19	7
	Propose a solution with a design document	19/11/19	49	30/11/19	11
Implementation of POC					
PAC3	Implement proposed design in Go and write PoC documentation	1/12/19	61	15/12/19	14
	Test PoC	16/12/19	76	19/12/19	3
Write TFG document					
	Write intro. + first theoretical part	4/10/19	4	31/10/19	27
	Write proposal and design doc part.	1/11/19	31	30/11/19	29
	Write analysis of implementation and conclusions	1/12/19	61	22/12/19	21
Delivery	Document review and video presentation	27/12/19	87	7/1/20	11

Figure 1. Tasks calendar.

This Gantt chart is the initial proposed set of tasks, which during the progress of he project might experience deviations, caused by wrong assumptions or lack of information at the time of writing.

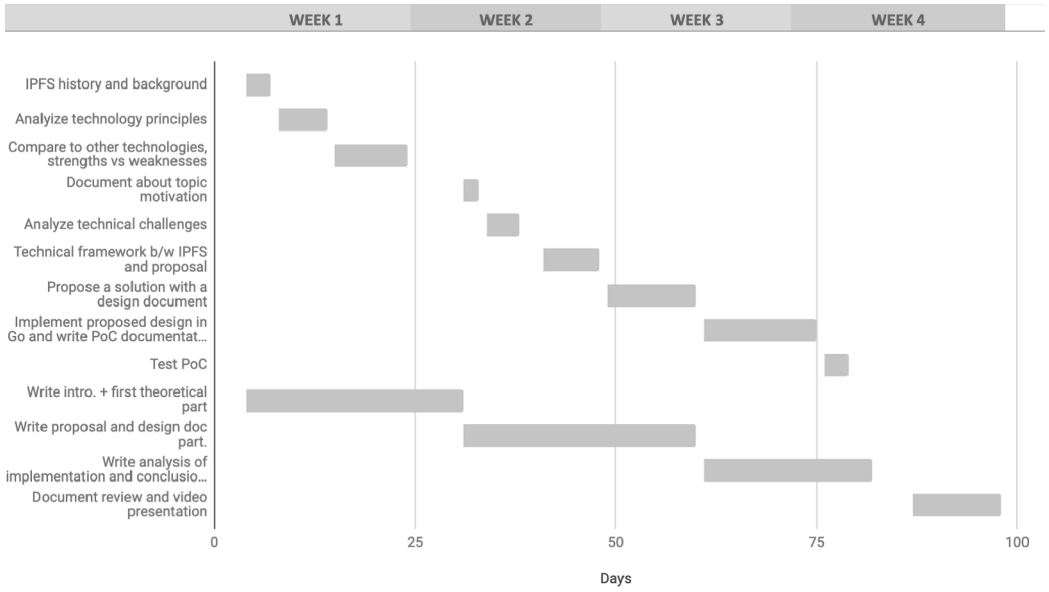

Figure 2. Gantt chart.

To minimize the impact of those deviations to the final project, we might end up reorganizing or re scoping some of the initially proposed tasks.

Any of those changes will be amended and commented in the final conclusions.

1.5. General vision of the solution proposed

We want to develop a distributed, anonymous file storage on top of IPFS. The scope of this work is to design the system and its API and to validate its viability in a real world environment by developing a PoC. Out of the scope is to create user faced applications or UI's that make use of the designed system.

In general lines the solution we proposed will:

1. Allow users to authenticate in an anonymous way to the system.
2. Grant access to users only to their files.
3. Allow users to upload and download.
4. Allow users to access their files via different devices.

This is the basic functionality the solution achieves to provide.

1.6. Chapters summary

1. ***Introduction***: in this chapter we do a general overview about the project goals, planification and execution plans.

2. ***Background: IPFS***: in this chapter, we make a deep dive in IPFS origins, technical background and ecosystem.

3. ***Other distributed file systems***: quick overview of two of the most relevant distributed file systems, to add extra context with IPFS.

4. ***Requirements and research questions***: visit to the requirements of our system, and presentation of some research questions we might find interesting to answer at the conclusion of this work.

5. ***Design of an Anonymous Decentralised File Storage***: in this chapter we will go through all the necessary phases required to elaborate a design document for our proposal. The goal of this chapter is that by the end of it, we will have a specification ready to be implemented.

6. ***The proof of concept*:** after having implemented our design, we visit its usage and structure, along with some end to end test to ensure it complies with our requirements.

7. ***Conclusions*:** wrap up of the work, visiting our research questions looking for answers, also evaluating the quality of the work, any future improvements or research areas that might be interesting for the future.

2. Background: IPFS

In this chapter, we do an overview of IPFS: how it originated, what technologies is built on top of, etc.

With this we want to give the necessary background to understand the motivations of this work and to present the required technical foundations to justify our proposed solution.

2.1. Brief history

IPFS paper [1] was first published in 2014. In it, Juan Benet describes the design of a distributed file system, that is content-addressed, verifiable, immutable and tolerant to failures, because all nodes act as equals and there is not a single point of failure.

His motivations for the design of such system were, in his own words:

... we are entering a new era of data distribution with new challenges: ... Many of these can be boiled down to "lots of data, accessible everywhere." Pressed by critical features and bandwidth concerns, we have already given up HTTP for different data distribution protocols. ***The next step is making them part of the Web itself.*** *(IPFS paper [1])*

So the challenge IPFS tries to overcome is to make it ***the Web itself***, and this is a very powerful concept. By having this distributed file system, to which anyone can upload files that are guaranteed to be immutable and versioned, we can ensure that the contents of the web are accessible for as many people as possible, for the maximum time possible, and without any third party, government or middleware tampering or censoring its contents.

At the time of presenting the paper, its author, Juan Benet, founded Protocol Labs, which is now the company leading the efforts of developing IPFS and its ecosystem. Even though Protocol Labs is a major contributor to the project, IPFS is an OSS project, and therefore many other organizations and individuals contribute to its development.

IPFS is still a project under development, but its first stable released was launched in February 2015 and is currently at version 0.4.22 at the time of writing (Oct 2019).

IPFS has defined a roadmap [3] towards version 1.0.0, which include work on authentication, anonymity and scalability as protagonists for next improvements.

2.2. Technology

IPFS is built putting together a combination of concepts and technical solutions that already existed. In this section we are going to explain some of those to give the required context to understand next ideas and references.

2.2.1. Distributed Hash Tables (DHT)

A Distributed Hash Table is a type of distributed system that allows for storage and retrieval of data associated to a key, in a similar way as a hash table [5]. The data stored can be of any kind, and it is associated to a unique key, used as an identifier across the network. Usually this key is generated from hashing the data with a hash function.

The network is made of peer nodes, and the responsibility of keeping the references to keys, and to store the data, is distributed among them. This way, the network achieves high availability and resilience to node churn and network partitions.

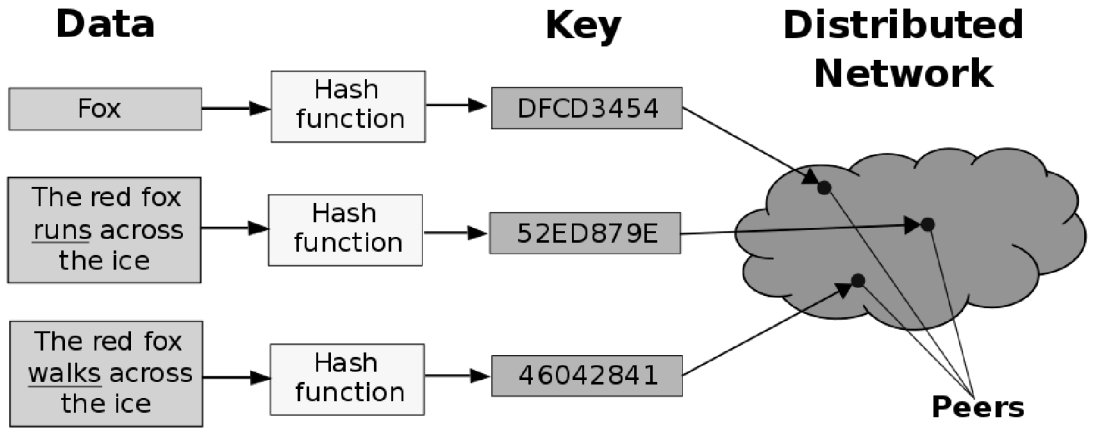

Figure 3. DHT representation.

Different designs of DHTs usually share a common set of characteristics between them, and that characterize the usual understanding of what a DHT is and how it works [5,6]:

Peer Discovery

Peer discovery is the process of locating nodes in the network when a new node joins or leaves, and also as peers come and go. For this, a list of nodes is kept up to date as peers come and go, all nodes keep a copy of this list, and they are assigned as bootstrap nodes dynamically for new peets to contact them, and therefore be able to acquire the list of other peers.

Scalability and Fault-tolerance

Since the network equally distributes the responsibility of storing and delivering routing information, and also of the distributed storage, DHTs scale well for large number of nodes. Some of the most popular implementations have hundreds of millions of active nodes at a given time.
For the same reason, DHTs are fault-tolerant to node churn, because the data is distributed across, the system can recover from faulty nodes.

Distributed Data Storage

The data is propagated and store to nodes that are closer from the key of that data. This is calculated using some distance function. This consideration makes for faster retrievals, since data related is held closer in node clusters.

Keyspace Partitioning

Most implementations use some consistent hashing or rendezvous hashing as their hashing functions. This is because both of them reallocate a minimum amount of keys on additions and removals of nodes. This tries to minimize the reallocation of data on node joining and churn.

Overlay network

Each node maintains a list of nodes to which it is related. This forms an overlay network and its topology will vary depending on the DHT implementation.

Most of the overlay networks topologies, aim for similar goals: to guarantee that the route length (number of hops) is low, so that requests complete quickly; and that maximum node degree of any node (number of neighbours) is low, so that maintenance overhead is not excessive. There are different choices and tradeoffs, but most implementations opt for solution that allow for more flexibility, and usually achieving $O(\log n)$ complexities for both constraints.

IPFS leverages concepts and ideas from several implementations of DHTs. It uses a variant of Kademlia DHT to replicate its content, and also adds concepts of BitTorrent reward system to encourage sharing of content vs leeching. The differences with Kademlia are mainly that IPFS maps Peer IDs to Content IDs and that the peers actually store information about where to physically locate a given piece of content [9].

2.2.2. Merkle Trees

A Merkle Tree [7] is a tree graph in which each leaf node has the hash of a data as the label, and each non-leaf node is labelled with the hash of all the hashes of their children nodes.

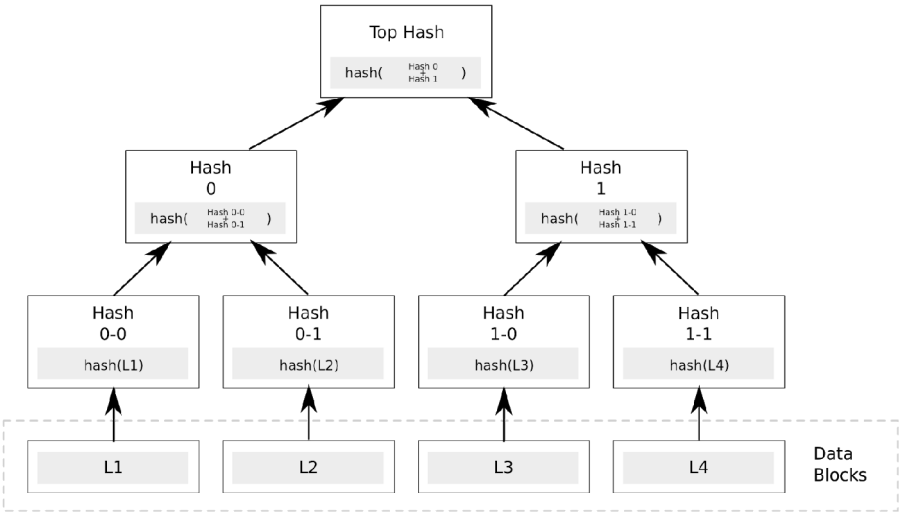

Figure 4. Merkle Tree

Merkle trees can be used to verify any kind of data that is stored or transferred between computers. Some variants of it are used in BitCoin, Git and other projects.

2.2.3. Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a graph in which, by traversing it through the edges, it is impossible to visit the same node more than once.

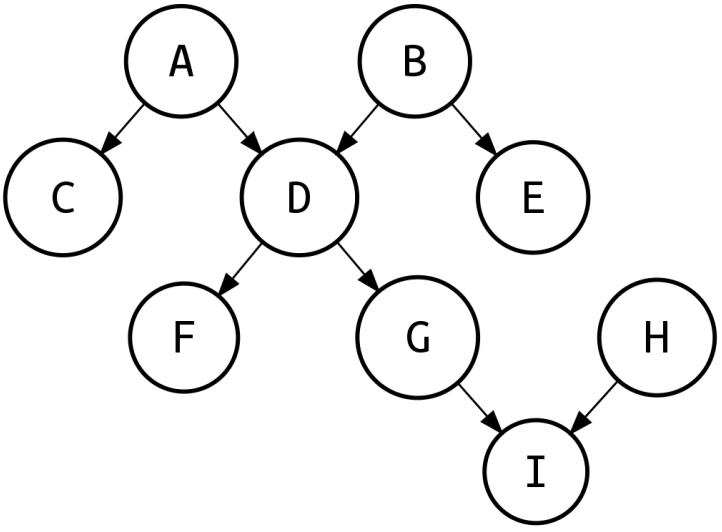

Figure 5. Directed Acyclic Graph.

That it is directed means that its edges can only be traversed in one direction, so it is not possible to go to the origin node from the source using the same edge.

By acyclic it indicates that there are no cycles in the graph, as in, as already said, is impossible to go back to an already visited node.

2.2.4. Merkle DAG

Merkle DAG [8] stands for Merkle Directed Acyclic Graph, and it diverges from a regular Merkle Tree in that it does not need to be balanced and its non-leaf nodes are also allowed to contain data.

It is a data structure that comes from the combination of concepts from Merkle Trees and Directed Acyclic Graphs.

IPFS uses Merkle DAG for content addressing, tamper resistance and deduplication. IPFS Merkle DAG is a generalization of Git's Merkle DAG implementation.

Precisely because Git uses a similar Merkle DAG, we can see that this data structure can be used for file versioning. Is for this reason that IPFS not only uses them for referencing data chunks for files, but also to keep a history of the files, so any one can visit any version of a given file, meaning that data in IPFS can be stored forever.

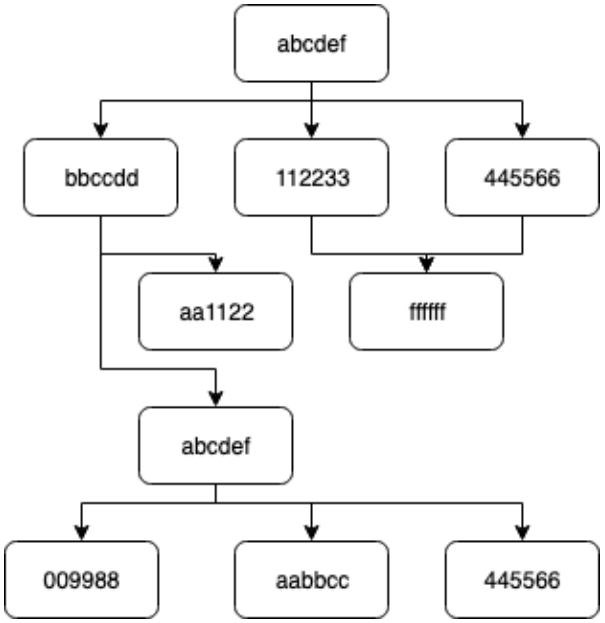

Figure 6. Merkle DAG.

2.2.5. BitTorrent

BitTorrent is a file-sharing peer-to-peer protocol that is widely used to distributed files across the Internet. At some point it reached 40% of the global internet daily traffic [12] with more than 150 million active users.

It has no search capabilities, this is why it uses .torrent files to identify what blocks of data it needs to fetch from peers. It is a peer to peer network built on the public internet, so the peer discovery is a challenge. It uses trackers to bootstrap peer connections and start communications between them.

It has some interesting algorithms to decide how and to whom share data blocks, and IPFS inspired on them to build its own block exchange BitSwap:

1. Firstly, it favors seeders versus leechers, rewarding peers that share a lot with priority when requesting files themselves.
2. It tries to keep rare blocks alive by sharing them first and to as many peers as possible.

As said, IPFS takes those principles and extends them to build BitSwap, which is a data block marketplace where nodes can exchange files.

2.2.6. Self-Certifying FileSystems

Self-Certifying FileSystems were described in David Mazièrez thesis at MIT in 2000 [10]. The paper describes a world wide file systems, to which anyone, from anywhere can access any published file, since all files share a common namespace. This file system would separate the file storage from the key management, making public keys part of the filesystem (and thus, self certifying).

This idea brings the system with much more versatility when it comes to how its users can authenticate the data contained in it, and since key management is aside from file storage, is trivial to make a single namespace with all files accessible.

IPFS brings this global addressable file system to live and takes some of its ideas and technical principles to build the system name service, called IPNS (InterPlanetary Name Service)

2.3. Components

Now we visited the main ideas IPFS builds on top of. IPFS itself is organized in several sub protocols or subsystems, each of which is influenced in one way or another by one or more of the previous technologies. We are not going to get in depth a lot in each of them, but to outline the main concepts and terms we need to be familiar within each one.

2.3.1. Identities

Nodes have a NodeId that identifies them. Nodes are granted with a public/private key pair to secure their communication with other nodes. The NodeId is the result of applying a hash[1] function to the public key, using a crypto puzzle that is the same as used in S/Kademlia [13] protocol.

Even though nodes can re-generate the NodeId at every startup, the system incentivizes them to remain the same, by granting long running nodes with some accrued benefits. Every time a node connects to each other they both check others public key by running the hash function, if it does not match, they can terminate the connection.

2.3.2. Network

Networking is a key component of any distributed system, this is why IPFS' networking subsystem is thought to provide both reliability and flexibility:

1. Is transport agnostic. Even though is best suited for WebRTC or uTP, it can work on top of any transport protocol, making it possible to even work on top of overlay networks.
2. Since is transport agnostic[2], it has reliability features, which make it suitable to use on top even of protocols that do not provide reliability guarantees.
3. Peer discovery is a key feature, this is why IPFS use several NAT traversal techniques to maximize connectivity even with peers inside firewalled networks.
4. It can validate messages using checksums.
5. It can sign messages using peers private keys.

2.3.3. Routing

The routing subsystem is in charge of finding other peers and find particular objects inside the peer swarm. IPFS uses a DSHT[3] (the S stands for Sloppy) which is based in S/Kademlia and Coral [14]. One of the particularities of IPFS implementation is that it makes distinction between values based on their size.

[1] IPFS makes use of multi-hashes, that are self describing values that tell you both the hash function and the value. This makes for easy interchangeability of hash functions in the future as better ones are discovered.

[2] Similarly to multi-hashes, IPFS also uses multi-address format to interchange peer addresses. These are self describing addresses, that bring in itselves all necessary information, from protocol to address.

[3] Different use cases can benefit from different implementations, IPFS exposes the Routing subsystem as an interface, this way it is easy to replace it as it fits, as far as the interface remains consistent.

2.3.4. Exchange

For exchanging data blocks, IPFS implemented BitSwap [1], its own protocol based on BitTorrent. It consists of a blocks marketplace, where nodes exchange data as value. The system needs to be as fair as possible, engaging sharing and penalizing leeching. There are scenarios where this is not possible, like when a node just joins the network. In those cases the node requesting data can work for the node it requests data from, looking for blocks of it interest, to gain benefits and do not be ignored by it.

In general the protocol needs to work for making the system as balanced as possible, with the data moving between nodes and do not create leeching nodes or nodes that are just seeding. BitSwap has some key components to pay attention to when achieving this:

1. Credit: the protocol keeps a credit balance of served/received. This balance might be in debt, and nodes tend to serve optimistically expecting to be paid later on. The probability of a node accepting to serve another is calculated using a function that takes the credit into account, so the more in debt a node is, the less probable any node want to serve it.
2. Strategy: this is an interface defining the function a node utilizes to choose to serve or not. The protocol uses sigmoids, but the team is exploring more. For example, strategies might include some sort of virtual currency., or others utilized in other protocols, like tit-for-tat from BitTorrent. In general any function will aim to maximize trade for the node, prevent leechers to operate, be unaltered by other unknown strategies, and be lenient on trusted peers.
3. Ledger: a history of exchanges is kept in each node to prevent tampering. In practice, is not required to operate, but a node can choose to identify this as an attempt to get rid of debt and stop the connection with that peer. More often, it is used as a proof that all history is tamper free, checked on both ends and if it is not the same, the connection ends. Old ledger information might be truncated anytime, since it probably belong to old nodes.

2.3.5. Objects

Previous subsystems build the peer-to-peer system, and on top of it, IPFS needs to store objects. For this, it implemented an Object Merkle DAG, which is a variation of Git's Merkle DAG, which give an interesting set of features such as: content addressing using multi-hashes as identifiers, verification by checksums which means is tamper free, and by design deduplication (if portions of different files generate the same chunk, the reference will be shared, since they are content addressed).

This data structure is so flexible that many other data structures can be modelled on top: linked lists, databases, blockchains, … giving a lot of flexibility to IPFS users.

There are some concepts that is useful to be familiar with for future reference:

1. Local objects: any object in IPFS ends up being stored in the local storage of some node. When a node requests some file, its blocks are downloaded from nodes that have them in their local storages and is copied to the node's own. To read a file it needs to be in our local storage (either in the filesystem, which is permanent, or in memory cache).
2. Pinning: files are copied into local storage when pinned. Pinning means storing permanently (or until unpinned) an object in a node's storage, and it can be performed recursively to download all children objects and links (a complete file versus a portion of it). This is why objects are permanent as far as any node have pinned them.
3. Publishing: the action of publishing an object will hash and split it into blocks, then telling other nodes it is available for them to fetch. Objects are idempotent, they never change. This is because IPFS is content addressed, so if the content changes, a new object is created which means any version can be fetched. Is important to note that when published, objects are not replicated. They will be replicated only when some other node tries to access them.
4. Object-level Cryptography: IPFS allows for cryptographically secure objects. Those are wrapped in a special construct that allows for encryption/decryption and verification.

2.3.6. Files

A versioned file system sits on top of the Merkle DAG, similar to Git. A file object is composed of several important parts to be familiar with:

1. Blob: a blob object is an addressable unit of data representing a file. It is similar to Git's blobs. They only hold data and no links or extra information is stored with them.
2. List: represents a large or deduplicated file. It holds a list of blobs or other lists (when deduplicated) that are portions of a big file.
3. Tree: a tree represents a directory, and holds no data, only a list of objects with names and references to any other kind of element: blobs, lists, trees or commits (see below).
4. Commits: likewise to Git, it represents a snapshot in the history of an object. An object can be of any kind. Comparing two commits contents give you with the differences between two versions. Git tooling can even be used with some configuration to explore IPFS file system, as the data models are very similar.

2.3.7. Naming

Since IPFS and its object storage are immutable, in order to access a file we can do it using its hash. What happens when the file is updated? We need the new hash, otherwise we will access the outdated version forever.

In order to prevent this, IPFS has a subsystem called IPNS, which targets making possible to have mutable pointers to immutable state. Those names, are Self certified names by the nodes. They sign the names at the time of publishing, so other nodes can trust their contents.

3. Other distributed file systems

There are other systems in use that might be relevant to our use case, either because they focus on anonymity or in access to private data. We will explain briefly two of the most important ones that focus on privacy and anonymity, which are main goals of our own proposal, to give more context about what IPFS offers versus other solutions.

3.1. Tahoe-LAFS

Tahoe-LAFS[30] stands for Tahoe Least Authority File Store, and it is a distributed and decentralized data store and file system. As its name makes reference to, it is based around the Least Authority principle, which basically means that any participant in the network only has the minimum set of privileges required for performing an operation.

This has differences with Google Drive or Dropbox do, since both providers hold access to all data stored on their systems, along with user information and other metadata, which they really do not require for only storing the files.

When it comes to topology, a Tahoe-LAFS setup is usually called a grid. Grids are made from three types of nodes:

- Introducers are in charge of putting new clients in contact with other nodes. If all introducers fail, current clients of the network will still work normally, but any new joiner will not be able to find any peer to connect to. Introducers are optional, since a network can also start if a list of storage servers is shared offline or pre set by default.
- Storage servers are the ones in charge of storing and distributing the data
- Client servers access the data held in storage servers. A single node can be at the same time a Storage and a Client server.

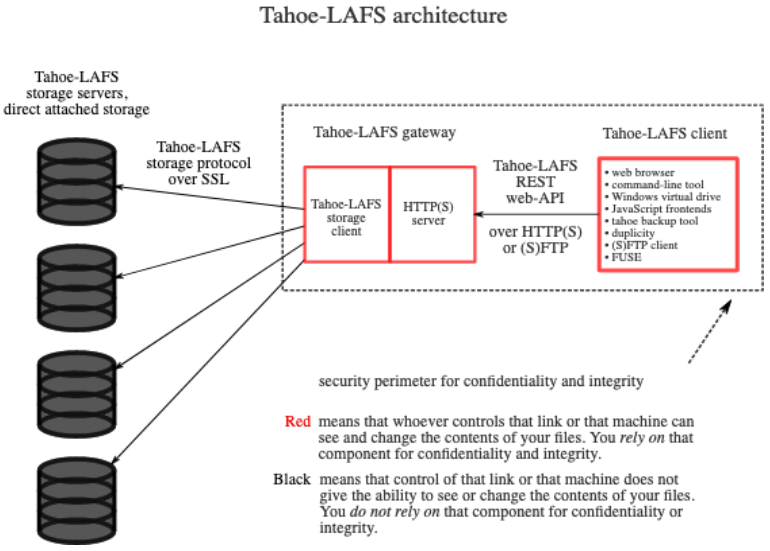

Figure 7. Tahoe-LAFS Architecture.

By its principle, Tahoe-LAFS encrypts all its files using a symmetric key, and after encrypting them files are chunked, similarly as in IPFS. This system though, has built in erasure coding[31], which allows to rebuild a chunked file with just a partial set of the total chunks, for example, if a file is split in 5 chunks, it is possible to rebuild it from any 3 of them. This trades storage capacity (erasure encoding needs more space than not having it) for reliability and error recovery.

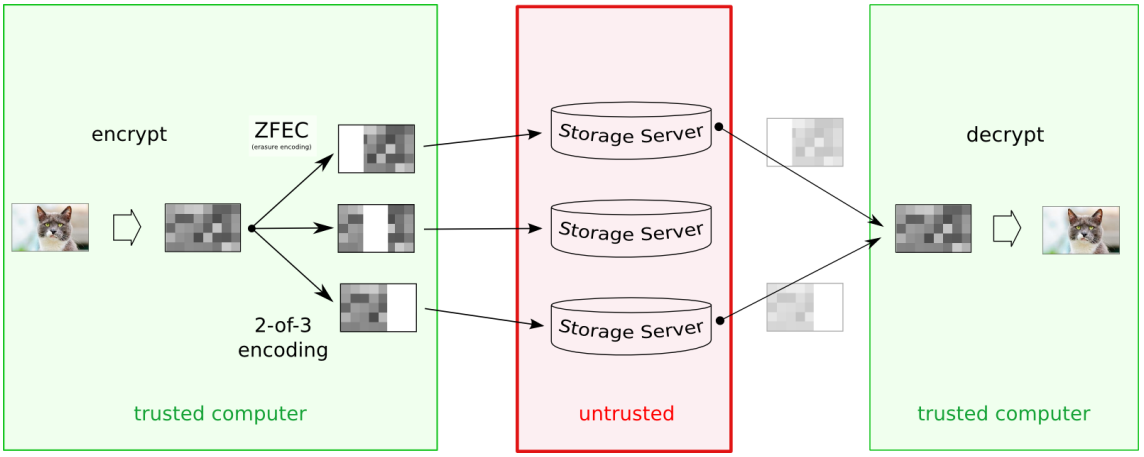

Figure 8. Erasure coding in Tahoe-LAFS.

Tahoe-LAFS works around the concept of capabilities[32], there are different set of access patterns to a file, that allow for different operations: verify capabilities allow for verification of file integrity, read capabilities for read access, and write capabilities for write access. Last ones are only possible for mutable files. This system let the data owner to share different access levels to each file, folder or subfolder, without compromising the data at any moment.

3.2. Freenet

Freenet[33] is a system similar to IPFS. It is a peer to peer network, decentralized and it was born to fight censorship.

Its design[34] separates the network from the access interface, for this reason there are several ways of accessing it. In this sense it differs from bittorrent and others, in such that them provide the way to access the files in the network embedded in the protocol, making it difficult to browse on it. Some popular ways of accessing the network are FProxy, which provides a browser-like experience.

Since the main goal of Freenet is anonymity, there are some operations that are performed differently than in other networks. When it comes to upload files, is very similar to what we want to achieve, it automatically distributes files, so the user who uploaded it can be immediately offline and their files will remain available.

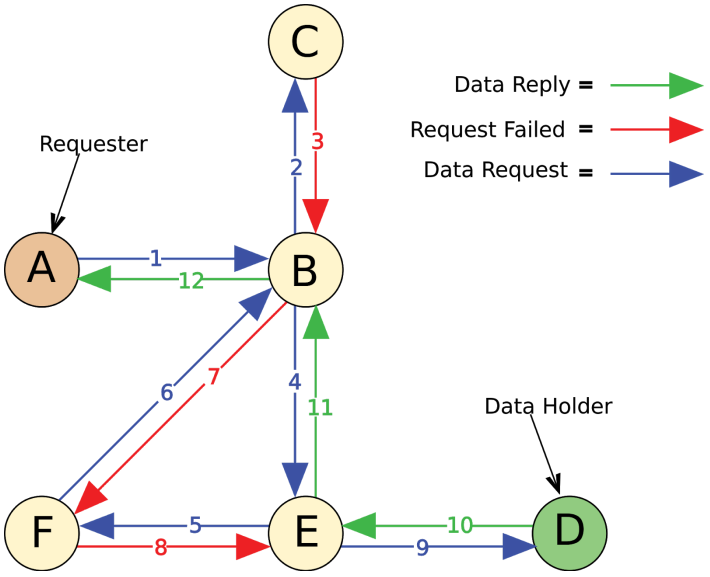

Figure 9. Request life cycle in Freenet.

Another difference, is that when a file is downloaded, the request is not directly made to the data holding nodes, but routed across the network, this is to increase anonymity of the requester. Usually this results in slower transfers, since bandwidth is very much spent on this extra round trips instead of just download throughput.

Since the main focus of Freenet is censorship and free speech, any document pushed is automatically made public, so there is no encryption by default of the contents or any control over who “owns” what. In the network all nodes are equals and have access to the same files.

4. Requirements and research questions

For this system to work, there are a minimum set of requirements it needs to fulfill to be able to work properly. Some of them are already covered by IPFS and others, which we might use directly or get inspiration from, while others will need to be built from scratch.

Requirements	IPFS	Tahoe-LAFS	Freenet
Decentralized	Yes	Yes	Yes
File encryption	Yes (under test)	Yes	No
File access authorization	No	Yes	No
Fully decentralized peer discovery	No	No	No
Automatic file replication	Yes (for cluster setups)	Yes	Yes
0 config NAT traversal (for easily startup and peer discovery over the Internet)	Yes (for non cluster setups)	No	Yes
Per user storage quota	No	No	No
Ability to create private networks	Yes (with cluster)	Yes	No
Extensible	Yes	Yes	Yes

Figure 10. Requirements comparison table.

As we can see there is no single of the visited systems that fulfills all our requirements. We also see that IPFS only fulfills some of them under a specific setup, this is something we will take into consideration when building the PoC.

Even though IPFS does not fulfill all of them, and might seem maybe others are a better starting point, we have chosen it mainly because it is the most active community wise, plus it is the most approachable to start developing with. We think development experience is a huge differentiator when it comes to adoption, and this is why we are going to focus on it.

After this visit to IPFS and others, we have now enough context to design the system proposed in Chapter 1.

Along the way we would like to present some research questions, regarding the solution but also IPFS and its ecosystem. We will try to answer them based on our results and the experience working with IPFS, and will present the conclusions for them at the end of this work.

5. Is it feasible, using existing components, to create a system such as the one proposed?
6. If a system like the presented exist, what would it take for its wide adoption?
7. How easy to work with is IPFS and its ecosystem when working to extend it?
8. What limitations were faced that were unexpected at the beginning of this work?

Answering the above questions, should give enough context and material for future works and research, and also give an idea of the current state of the development experience for the IPFS projects.

5. Design of an Anonymous Decentralised File Storage

After setting up the required technical background, in this chapter we are going to present what the solution we pretend to build looks like. To do so we will follow a set of steps:

1. Present a high level picture of the required components and features we want the system to have.
2. Put them in context with what IPFS and others have to offer.
3. Analyse any limitations or challenges the project might suffer because of IPFS.
4. Explore IPFS ecosystem for solutions to those possible challenges, or propose custom built solutions to overcome them.

5.1. Scoping

We already introduced the scope of the system in Chapter 1, but we will revisit it for reference:

1. Be built leveraging IPFS platform.
2. Allow users to authenticate in an anonymous way to the system.
3. Grant access to users only to their files.
4. Allow users to upload and download files.
5. Allow users to access their files via different devices.

The components required will be, roughly, the following, from the inner layer to the outer one of the system:

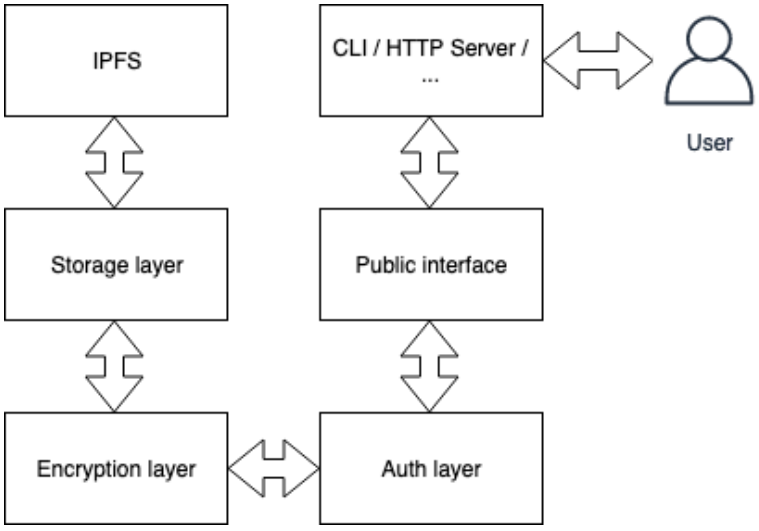

Figure 11. Components schema.

A **storage component** which will be in charge of publishing and managing the files in IPFS.

An **encryption component** that takes care of encrypting and decrypting user files so they can be read and safely stored.

An **authentication and authorization** component in charge of identifying users and granting only access to their files.

An **interface** exposing this subsystems so the system can be used as a library by other developers.

In this work we will develop **a CLI** to interact with the system, but other options would be also good to interact with the interface, such as an HTTP API or a gRPC one.

5.2. Technical challenges

With the above requirements, we can see some technical challenges we will need to overcome in some areas:

5.2.1. Authentication

Our system requires that a use can authenticate in it, and therefore only access their own files. IPFS has no authorization or authentication mechanisms, even though it provides some encryption capabilities, is the responsibility of the user to know what files to fetch. We want our users to be able to install a fresh copy of the application, authenticate, and the system will then handle the fetch and decrypt of user files on its own.

This is a challenge in itself, and we will need to investigate solutions to workaround this.

5.2.2. Distribution of the files

IPFS does not distribute the files automatically, this is an issue for our intentions, since we want that any file a user publishes to be automatically saved in the swarm.

There are no means to force remote nodes to pull our data, so there is no possible workaround to use the public IPFS network.

There is a parallel project, called IPFS Cluster [15] that allows users to build a private IPFS network, with custom replication factors.

We could leverage this to spawn an IPFS Cluster node with each one of our application instances, creating a swarm between our users. This brings the next issues.

5.2.3. Peer discovery

Public IPFS network has centralized, public bootstrap nodes maintained by Protocol Labs and others. This makes easy for new joiners to the network to start communicating with others.

If we use IPFS Cluster, we can’t make use of this approach to discover peers. So we need to find out a way to discover peers when joining and to keep an up to date list of valid and stale addresses.

5.2.4. Storage appropriation

By default IPFS Cluster pins all files in all nodes. This is desirable in a private network environment, but for us is the opposite.

A good approach to make the system appealing to use, would be to let people store in the device the same amount of data they lend to the system to store others data. So if I have 100MB uploaded, I need to let the system store other 100MB, so in order to make use of the system users pay in storage. That system is not perfect but for the PoC we will go with it.

In any case there is no way to reserve a given amount of disk space to others' data, and we need to make the system aware of this if possible, being this possibly one of the trickiest things to solve of the list.

5.3. Design of the components

5.3.1. Network communication

We need to isolate ourselves from the main IPFS network. We do this by setting a different network ID and by removing all bootstrap servers from our setup. We will also use IPFS Cluster with a specific key, which as said previously lets us leverage forced file replication.

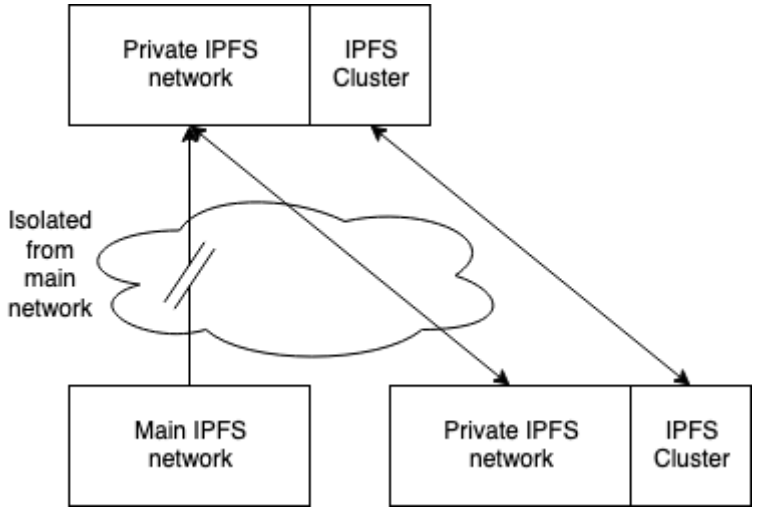

Figure 12. Network setup.

5.3.2. Authentication

As said before, the system needs to let users authenticate and only provide access to their files in the system, and not to other users' files. This is problematic because in IPFS there is no such concept of a user or authority.

For such reason, we will explore a similar solution as Bitcoin[16] uses, that is not authenticating using a user and password mechanism, but an encryption key. Any user that have the encryption key used to upload a given set of files, will be able to add more and access the said set.

As a result of this, both the encryption and authentication components are highly related in the system.

The mentioned key consists of an AES256[17] key that is generated when the app configuration is initialized. This will be used as a unique identifier for its user. This means that if the user loses the key will lose access to any file that they have uploaded previously using it.

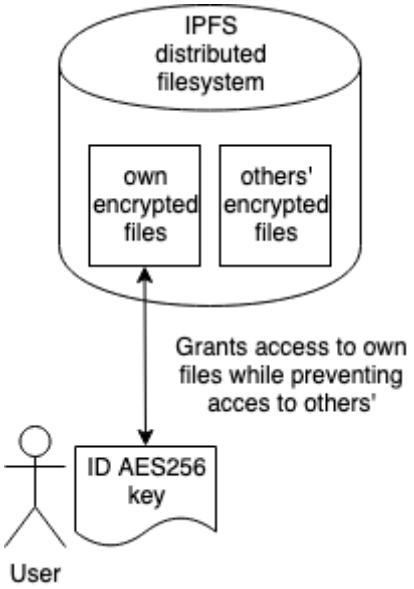

Figure 13. Authentication.

As an upside, that key can be stored securely in any offline device, and copied back into any other computer, meaning that the files can be accessed and recovered as far as the is key is correct.

5.3.3. Peer discovery

As a main goal, we have decentralization. This includes peer discovery.

For peer discovery in a P2P network a given node needs access to at least one other's address, so they both can share other peers addresses and grow their list of connected peers. This means that if a node is not provided with a list of peers to bootstrap from, there need to be any other mechanism to get them.

In IPFS there are bootstrap nodes, which are controlled by Protocol Labs and are used to provide a central point of reference when a node joins the network.

For us this is not desirable. First because we do not want our system to be part of the main IPFS network, second because relying on mechanisms like that defeats some of the purposes we try to achieve.

On the other hand, IPFS Cluster provides mDNS peer discovery for local networks, but this means it has no means of crossing network boundaries by its design. We could try to bypass that by setting up a VPN to join to and have special configuration to redirect multicast traffic through it, but again, a VPN needs to belong to a known party to be considered secure enough and adds centralization to the system.

We are not the first ones facing this problem, for example, Bitcoin faced it also on its beginning and solved it mainly in three ways:

- The aforementioned bootstrap servers
- You could provide manually a list of peers where to connect if you knew and trusted them.
- They had an IRC peer discovery implementation, which is now abandoned.[19]

The third option is what we are going to implement. Let's see some of their advantages:

- Fully decentralized: there is no need for centrally controlled nodes that are part of the network, just to join any publicly available IRC network, maybe even more than one for reliability.
- Bypasses NAT, since the nodes are the ones communicating with the IRC servers, is less likely to face NAT problems for peer discovery phases.
- Using IRC as a protocol lets for ease of implementation, since it is a very simple and old protocol with wide support across many platforms.

All these do not come on their own, since there are also disadvantages to its use, and this is why Bitcoin dropped this mechanism:

- Slow. IRC is heavily based on text operations, meaning it is slow, also since it is not a socket based communication between peers, some turn-to-speak protocol needs to be in place, hence is easy to lose connections, and adds a significant network overload to keep the node up to date with the peers in IRC.
- Scales up to a certain point, after some number of nodes joins a given IRC server, the server is going to suffer the extra load, and is possible that servers are not willing to take that cost on their side. This was an important factor when Bitcoin decided to drop

support for it, since Freenode[20] kicked them from their network because was affecting the entire user base.

For our design those are not really problems, so we can go with it as an initial approach to the problem. Our design is going to consist of different phases:

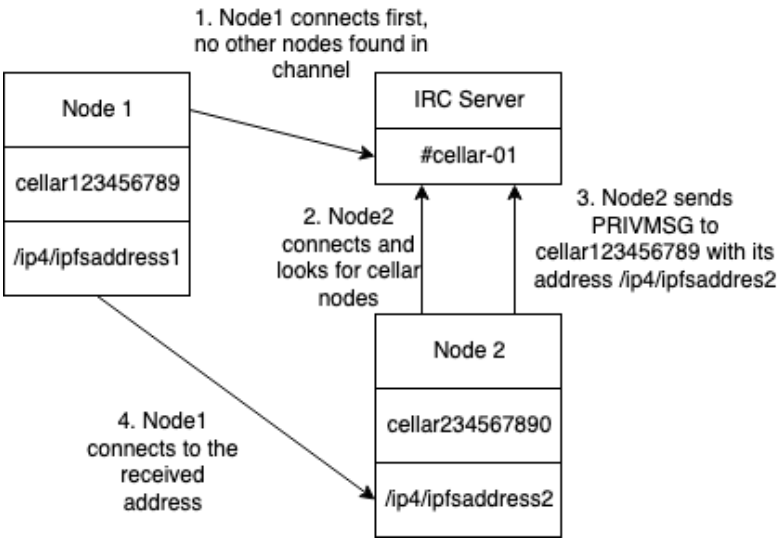

Figure 14. Peer discovery life cycle.

- When a node starts, will join a specific IRC server, with a random nickname that looks like `cellar123456789` where the suffix number is a random 9 digits number.
- After connecting successfully, will join a channel with a name like `cellar-1`, in this case the trailing number makes reference to the ISO[21] week number of the current day.
- After joining successfully, will look for other nicknames belonging to the application, and if any, will start a conversation with them, and sending its own address to each of them.
- Nodes on the receiving side, will take the received address and add it to their peers list, establishing a connection.
- All this process is going to take place periodically, to keep peers lists up to date.

5.3.4. Encryption

As we mentioned when talking about authentication, a unique AES256 key is used to identify who can access a particular set of files. We also mentioned that both, authentication and file encryption would be pretty much entangles subsystems because of this.

The process done when a file is added using our application is as follows. When adding a file:

- The original pathfile is given to the application
- It is then encrypted using the AES256 key into a temporary file
- That temporary encrypted file will be the one added to IPFS

When retrieving a file:

- The file is unencrypted on the fly and stored in the application file path

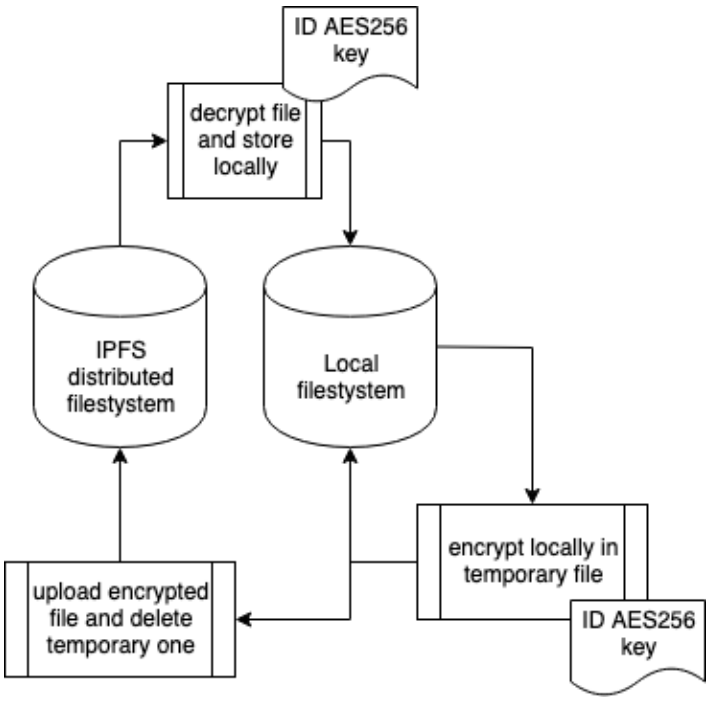

Figure 15. Encryption of a file.

Some advantages and disadvantages come from this process: on the up side it is difficult to infer any file contents, since even if two users upload the same file to the network, since both will be encrypted using different keys, their hashes won't match, so an attacker would have difficulties when trying to know what files a particular user has. As a downside, the space usage is much less efficient, since object reutilization is minimized because of this very same reason.

5.3.5. File discoverability and sync

As mentioned before, the only thing a user needs to identify themselves is the AES256 key, and also all files that are uploaded to the network are encrypted. The application needs some way to track what files belong to what user, both for user experience purposes (know the original file name and path) but also for synchronization purposes (under a fresh start, how do I know what files I need to download?).

IPFS creates an ed25519[22] key for each node. This key consists of both a private and a public keys. These key pairs are used for node identification during IPFS normal operation, and are replaceable without any implication for the node functionality. The same keys are used by default when publishing to IPNS and are used as the name of the node.

As we explained in previous chapters, IPNS is the IPFS name server, and it lets an IPFS node to publish, under a specific address, an IPFS file. The difference with any other IPFS file is that it can change the file it references to. We want to make use of this feature, and hold in IPFS itself, a list of files we need to download for a specific AES256 key.

As we said, the keys used to generate IPNS addresses are the auto generated ed25519 by default, but to be able to get back the published list of files, we need to make them reproducible. This is the reason why we generate a new ed25519 key pair [23] seeded with our AES256 identity, which let us publish under the same IPNS address, and also let the node maintain its

own address, which if we just copied the node identity over, we could not have multiple nodes accessing the same address hence they would collide.

The process of syncing files and discovering the list for a specific user is as follows:

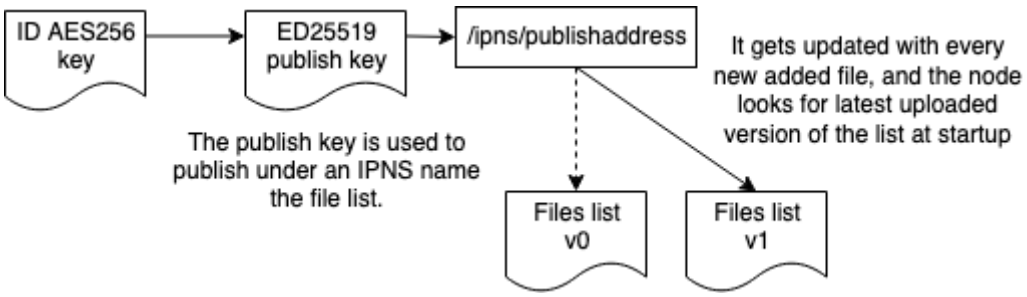

Figure 16. Keeping track of files.

A file with a list of added files for the user is kept locally. The file follows the following structure:

```
4
/folder/file1 QmZiWM4HUXuywMUVoAoymwiqk5ePD2FhgY1CDrbfDw9X6Q
/file2.png QmdyCWdkykp6ejwxdEkKUhb27SvXFWqYghWXjegjYNLRL7
```

Figure 17. Store.list contents.

The first line holds the version number, this number is incremented every time the file is updated, the following lines are a list of file paths mapped to their IPFS hash. This file is kept locally and is encrypted. Every time a file is added, this file is updated and published to IPNS using the publish key mentioned before.

Before uploading, we check that the version already published, if any, is more up to date than ours, in such case we merge both files and publish an update. What follows is a download and decrypt of any missing file to the local store. This way we maintain any number of computer running a same id up to date.

6. The proof of concept

We have implemented a proof of concept of the previous designed components and we will test it in this chapter.

Before jumping to test it, we will explain how it is structured and how each component interacts and behaves.

6.1. Overview

The proof of concept is built in Go language[24], mainly because it is the language both IPFS and IPFS Cluster are built in, and in case we need to reuse some of their components directly we would have a better time doing it.

6.1.1. The application

The application for this PoC is called cellar, it consists of a cli command which lets you add files and sync them in other computers and also to spawn a new cellar daemon, which is in charge of managing config and daemon lifecycle for IPFS and IPFS Cluster as required.

It creates a .cellar folder in your $HOME path by default. Inside this folder, there are configuration folders for IPFS and IPFS Cluster specially crafted to work as a private network, along with a keystore folder holding your ID key and a files folder, where files are stored for local access.

To use the application, first you need to run the init command, this will initialize all the configuration required to run the daemon, and create a new ID. If you already are in possession of an ID, is after this step when you need to place it inside ~/.cellar/keystore folder.

```
# cellar-service init

        You created a new cellar config. Your ID file is at
/Users/marcguasch/.cellar/keystore/id.key
     Please store this file cautiously, as if you start cellar in any other
machine you will need it to access your files.
```

Figure 18. Cellar init.

After this we can run the daemon:

```
# cellar-service daemon
Initializing daemon...
Repo version: 7
System version: amd64/darwin
Golang version: go1.12.7
12:31:19.014  INFO  service: Initializing. For verbose output run with "-l
debug". Please wait... daemon.go:46
Swarm is limited to private network of peers with the swarm key
Swarm key fingerprint: 9b5acbcb27d76fc16f961da690df1fc3
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/192.168.1.38/tcp/4001
Swarm listening on /ip6/::1/tcp/4001
Swarm listening on /p2p-circuit
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/192.168.1.38/tcp/4001
Swarm announcing /ip6/::1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
12:31:19.522  INFO  cluster: IPFS Cluster
v0.11.0+git5258a4d428600976ebae1b14be9205dfacdca920 listening on:

/ip4/127.0.0.1/tcp/9096/p2p/12D3KooWQea1Y8xMGD4LTdagGtG1iAz5q3qAJEv86VwRyQje
z4p1

/ip4/192.168.1.38/tcp/9096/p2p/12D3KooWQea1Y8xMGD4LTdagGtG1iAz5q3qAJEv86VwRy
Qjez4p1

/ip4/127.0.0.1/udp/9096/quic/p2p/12D3KooWQea1Y8xMGD4LTdagGtG1iAz5q3qAJEv86Vw
RyQjez4p1

/ip4/192.168.1.38/udp/9096/quic/p2p/12D3KooWQea1Y8xMGD4LTdagGtG1iAz5q3qAJEv8
6VwRyQjez4p1

12:31:19.525  INFO  restapi: REST API (HTTP): /ip4/127.0.0.1/tcp/9094
restapi.go:502
12:31:19.526  INFO  ipfsproxy: IPFS Proxy: /ip4/127.0.0.1/tcp/9095 ->
/ip4/127.0.0.1/tcp/5001 ipfsproxy.go:307
12:31:19.527  INFO         crdt: crdt Datastore created. Number of heads: 0.
Current max-height: 0 crdt.go:262
12:31:19.527  INFO         crdt: 'trust all' mode enabled. Any peer in the
cluster can modify the pinset. consensus.go:263
12:31:19.528  INFO  cluster: Cluster Peers (without including ourselves):
cluster.go:634
12:31:19.528  INFO  cluster:     - No other peers cluster.go:636
12:31:19.528  INFO  cluster: ** IPFS Cluster is READY ** cluster.go:649
```

Figure 19. Cellar daemon run.

Now we can add files to the store and will be synced across any other node in the cluster, if a folder is passed, will be added recursively:

```
cellar-service add README.md
ipfs -c /Users/marcguasch/.cellar/ipfs add README.md.ciph
ipfs-cluster-ctl pin add QmaNv278WFVC8ieyVG78jMX8SZpD1QrLvfqXrFddNUXQse
```

Figure 20. Cellar add file.

All files will be synced in ~/.cellar/files.

6.1.2. Code structure

The code is structured in various Go packages as follows:

- cmd: holds entry points for the cli commands, in our case the application is controlled from the cli with a cellar-service command, and this package holds the entry point for it along with util and helpers.
- config: holds methods to initialize and access config files, here is where the ID key is generated on init.
- crypto: holds methods to cipher and decipher files
- storage: holds methods to add and update files, also updates the store file
- The root package holds Cellar type, which is the daemon orchestrator. Is in charge of peer discovery and files synchronization over time.

6.2. Testing

To ensure our application works as expected, we set up a cluster with 4 nodes. This cluster will make a private IPFS/IPFS Cluster network, and there are several things we want to test:

- Peer discovery works as expected, peers connect to IRC and communicate with each other establishing the required connections, keeps working when nodes go down and when new nodes join.
- Files are added successfully, they are accessible through IPFS but encrypted, and synced automatically to the local storage, also that nodes can only access their own files.
- We want to ensure that an ID is safe to move along other computers and that its files are going to be synced automatically.

We will use Docker[25] for our tests. Docker is capable of running containers[26], which are linux kernel namespaces[27], making for lightweight virtualization if we want to compare with regular VMs[28]. With another tool, docker-compose[29] we are capable of orchestrate and set up container networks easily.

First, we are going to start the four nodes:

```
 # docker-compose build node1 node-test1 >/dev/null
Building node1
Building node-test1
 # docker-compose up -d --force-recreate node-test1 node1 node2 node3
Creating network "cellar_default" with the default driver
Creating cellar_node3_1    ... done
Creating cellar_node2_1    ... done
Creating cellar_node-test1_1 ... done
Creating cellar_node1_1    ... done
```

Figure 21. Starting the test cluster.

After some time, we see that they joined IRC and are communicating their addresses:

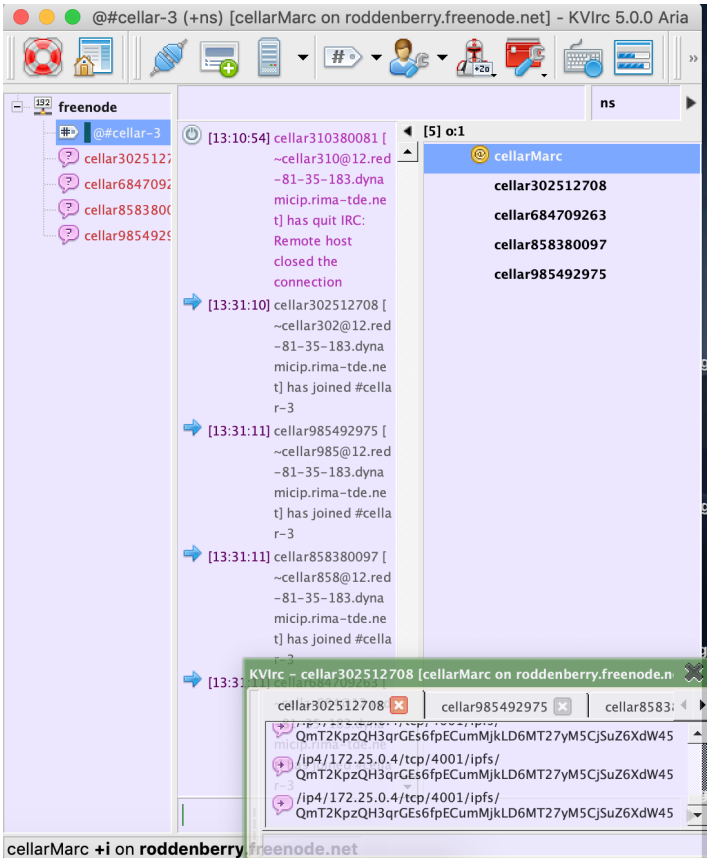

Figure 22. Nodes communicating through IRC.

We then see in the logs they are successfully establishing connections:

```
 # docker-compose logs
...
node3_1| connecting to
/ip4/172.25.0.5/tcp/4001/ipfs/QmdfUd1178vuTvzisdP6ocV8BnmD9R8ndkvikbepo17w1H
node1_1| connecting to
/ip4/172.25.0.5/tcp/4001/ipfs/QmdfUd1178vuTvzisdP6ocV8BnmD9R8ndkvikbepo17w1H
node2_1| connecting to
/ip4/172.25.0.5/tcp/4001/ipfs/QmdfUd1178vuTvzisdP6ocV8BnmD9R8ndkvikbepo17w1H
node1_1| connect QmdfUd1178vuTvzisdP6ocV8BnmD9R8ndkvikbepo17w1H success
node2_1| connect QmdfUd1178vuTvzisdP6ocV8BnmD9R8ndkvikbepo17w1H success
node3_1| connect QmdfUd1178vuTvzisdP6ocV8BnmD9R8ndkvikbepo17w1H success
```

Figure 23. Connecting nodes to each other logs.

Now we will connect to a particular node, and add some files in there, those files should be replicated to the other nodes, but never synced, since they belong to a different ID:

```
 # docker-compose exec node-test1 bash
root@7d32b85a056a:/app# mkdir -p ./level1/level2/level3
root@7d32b85a056a:/app# echo in1 > ./level1/in1.txt
root@7d32b85a056a:/app# echo in2 > ./level1/level2/in2.txt
root@7d32b85a056a:/app# echo in3 > ./level1/level2/level3/in3.txt
root@7d32b85a056a:/app# ./cellar-service add ./level1
ipfs -c /root/.cellar/ipfs add level1/in1.txt.ciph
ipfs -c /root/.cellar/ipfs add level1/level2/in2.txt.ciph
ipfs -c /root/.cellar/ipfs add level1/level2/level3/in3.txt.ciph
ipfs-cluster-ctl pin add QmVCJZRqYHmG8koidyeQqkh6vUtgta6UXGjCQUrihpoXfV
ipfs-cluster-ctl pin add QmeAU2RdBefiZ7FWVBRoaNoDegEQ7cDAFvJGVBchKNz8yo
ipfs-cluster-ctl pin add QmY4gNer3Wmhobic47mqJ1cJQkszt1akjbTC12MGzzvS1h
root@7d32b85a056a:/app#
```

Figure 24. Adding files in the test node.

Now see that files were updated, synced and the new list was published in the logs:

```
node-test1_1  | 2020/01/19 12:40:46 updating from 0 to 3
node-test1_1  | 2020/01/19 12:40:46 syncing /level1/in1.txt :
QmVCJZRqYHmG8koidyeQqkh6vUtgta6UXGjCQUrihpoXfV
node-test1_1  | ipfs -c /root/.cellar/ipfs cat
QmVCJZRqYHmG8koidyeQqkh6vUtgta6UXGjCQUrihpoXfV
node-test1_1  | 2020/01/19 12:40:47 syncing /level1/level2/in2.txt :
QmeAU2RdBefiZ7FWVBRoaNoDegEQ7cDAFvJGVBchKNz8yo
node-test1_1  | ipfs -c /root/.cellar/ipfs cat
QmeAU2RdBefiZ7FWVBRoaNoDegEQ7cDAFvJGVBchKNz8yo
node-test1_1  | 2020/01/19 12:40:47 syncing /level1/level2/level3/in3.txt :
QmY4gNer3Wmhobic47mqJ1cJQkszt1akjbTC12MGzzvS1h
node-test1_1  | ipfs -c /root/.cellar/ipfs cat
QmY4gNer3Wmhobic47mqJ1cJQkszt1akjbTC12MGzzvS1h
node-test1_1  | ipfs -c /root/.cellar/ipfs add /root/.cellar/store.list.ciph
node-test1_1  | ipfs-cluster-ctl pin add
QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq
node-test1_1  | 12:40:47.255  INFO      cluster: pinning
QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq everywhere: cluster.go:1398
node-test1_1  | 12:40:47.258  INFO      crdt: new pin added:
QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq consensus.go:209
node-test1_1  | 12:40:47.264  INFO      crdt: replacing DAG head:
QmdVX2wJ7kzABYt8Y8Wz9FKVfwVfVTUEQuyiNnoDDV1bdE ->
QmebfApfmrq3ivCe31J5Y8snU5AhLuxqg88CiWjpg5Yj3D (new height: 4) heads.go:82
node-test1_1  | 12:40:47.268  INFO restapilog: 127.0.0.1 - -
[19/Jan/2020:12:40:47 +0000] "POST
/pins/ipfs/QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq?name=&replication-
max=0&replication-min=0&shard-size=0&user-allocations= HTTP/1.1" 200 293
node-test1_1  |  restapi.go:117
node-test1_1  | 12:40:48.285  INFO restapilog: 127.0.0.1 - -
[19/Jan/2020:12:40:48 +0000] "GET
/pins/QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq?local=false HTTP/1.1"
200 1188
node-test1_1  |  restapi.go:117
node-test1_1  | 2020/01/19 12:40:48 Published to
12D3KooWNyS7BBKzFd6bwQRFXKmfHWJgDbJV6wyT5k7RrijHVoc3:
/ipfs/QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq
```

Figure 25. Files synced locally logs.

We also see replication happening in other nodes:

```
node1_1      | 12:40:47.310  INFO      crdt: new pin added:
QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq consensus.go:209
node1_1      | 12:40:47.327  INFO      crdt: replacing DAG head:
QmdVX2wJ7kzABYt8Y8Wz9FKVfwVfVTUEQuyiNnoDDV1bdE ->
QmebfApfmrq3ivCe31J5Y8snU5AhLuxqg88CiWjpg5Yj3D (new height: 4) heads.go:82
node1_1      | 12:40:47.394  INFO   ipfshttp: IPFS Pin request succeeded:
QmX2Tu8Mqdxv8tKD7vm9T9GnQgoPFdpzdoNFAJV6cZRLKq ipfshttp.go:372
```

Figure 26. Files sync in remote node log.

Now we need to check that in those nodes there are no synced unencrypted files:

```
docker-compose exec node1 bash
root@6a2165bca367:/app# ls ~/.cellar/files
root@6a2165bca367:/app# ipfs -c $HOME/.cellar/ipfs cat
QmVCJZRqYHmG8koidyeQqkh6vUtgta6UXGjCQUrihpoXfV | tr -d '\0'

root@6a2165bca367:/app# ipfs -c $HOME/.cellar/ipfs cat
QmeAU2RdBefiZ7FWVBRoaNoDegEQ7cDAFvJGVBchKNz8yo | tr -d '\0'






d?*k  i
root@6a2165bca367:/app# ipfs -c $HOME/.cellar/ipfs cat
QmY4gNer3Wmhobic47mqJ1cJQkszt1akjbTC12MGzzvS1h| tr -d '\0'
 +□  " 
```

Figure 27. Check files are encrypted in remote node.

As we can see, the other nodes do not have any file in sync locally, but have the pins matching our uploaded encrypted files, and their contents are in fact, encrypted.

Finally, we shut down node-test1, and start node-test2, which is a different node, that shares its ID. This should force previously uploaded files to be synced and visible in this new node.

```
 # docker-compose stop node-test1
Stopping cellar_node-test1_1 ... done
 # docker-compose up -d --force-recreate node-test2
Creating cellar_node-test2_1 ... done
 # docker-compose exec node-test2 bash
root@2e04b7447f22:/app# cat $HOME/.cellar/files/level1/in1.txt
in1
root@2e04b7447f22:/app# cat $HOME/.cellar/files/level1/level2/in2.txt
in2
root@2e04b7447f22:/app# cat $HOME/.cellar/files/level1/level2/level3/in3.txt
in3
```

Figure 28. Check files are sync on new node with same ID.

As we see, the files were synced, and out system is working as expected.

An automated version of this test is included in the code, which can be ran by typing ./test.sh in the command shell. A recorded execution of this automated tests is included in the work.

7. Conclusions

7.1. Answering the research questions

In chapter 4, we presented a set of research questions we wanted to answer. After working with IPFS and IPFS Cluster while developing our own solution, we are in a good position to answer them.

Q: Is it feasible, using existing components, to create a system such as the one proposed?

A: It is, in fact, our proof of concept accomplishes the majority of requirements stated at the beginning of this work. With some more resources, we could either fork IPFS and IPFS Cluster for a much more integrated solution, or we could extend them further and have a full system built on top transparently.

Q: If a system like the presented exist, what would it take for its wide adoption?

A: Tahoe-LAFS, as visited before, could be considered an existing solution that is very much like ours. Its adoption is niche though, and it is likely that even if our solution is full featured and production ready, it happens the same. This is partially because the know how required to approach them is high, and also because for an uneducated in CS end user, their benefits are limited and non evident. This means that a way of increase adoption would have two vectors: first, make them easy to use, drop in replacements for existing solutions. This means that if an end user could with no effort opt for registering with Dropbox and use its application, or to download something else, with added benefits and free without added friction, is more likely people would use it. Secondly, educating users in the importance of online anonymity, and the pitfalls of centralized, corporate controlled solutions, which makes their data a commodity. By educating them is likely that users would do more sensible choices when evaluating different providers.

Q: How easy to work with is IPFS and its ecosystem when working to extend it?

A: For the development of cellar, our application, we had to go through an understanding of various IPFS projects and codebases. Some components are thought as shared libraries to be used by third parties, like the go-ipfs projects, while others are not, or marginally documented even inside their own projects. This made for a hard time integrating natively. At the beginning, the choice for IPFS was more about the community and the project being maintained, because even though many features were useful and the project in itself is powerful and interesting. Those came with expectations of high extensibility of its components and project. At the end, maybe modifying Tahoe-LAFS would have been a better decision, since it is built on pillars that are more meaningful to us (capabilities, encryption by default, erasure coding, …), and IPFS ended up requiring a lot of discovering, code digging and troubles navigating vaguely documented features. Cellar, instead of being an integrated application with IPFS and IPFS Clusters API's and libraries, is a thin wrapper on top of their CLIs. Even if functional, this is far from ideal if we look for long term maintainability. If had to chose now, we would go for exploring alternatives, maybe extending Tahoe-LAFS or any other project.

Q: What limitations were faced that were unexpected at the beginning of this work?

A: As said in the previous paragraph, the main limitation was navigating the IPFS code projects, which are vaguely documented and require a lot of context that is difficult to get. On top of that, there were some topics we were not able to overcome, for example storage appropriation, which means disk usage is unlimited right now, insead of being able to set a budget per device, among others. But definitely the biggest one was that the IPFS project documentation and contribution process is difficult to navigate and get insights from.

6.2. Goals achievement

We will now do an overview retrospective about the accomplishment of the work goals. As stated at the beginning, we wanted to implement a distributed, anonymous file storage on top of IPFS that was easy to use and extensible. On this matter the proof of concept let us upload and sync files to and from an IPFS private network, so functionally our goals are achieved. On the other hand, the system is not easy to use from an end user standpoint, this is mainly because the way authorization works.

We failed at finding a way of authorizing users in an easy and transferable manner across devices. Even though carrying an ID file across devices might sound straightforward for some, truth is this is probably the bit that would cause more friction as a generalist approach, letting out many people. On that matter, we failed at making an accessible design. On the extensible side, we are again limited by the poor documented IPFS Cluster documentation, which ideally would be forked and be a fully start point for a new system, but its code base does not really make it simple, so more work is required there, either on this direction or finding another starting point like Tahoe-LAFS, for example.

6.3. Planification

When it comes to the planification, at the beginning we started following it with no issues, but once had to start developing our solution, the issues faced let us behind on some matters. Is because of that that the requirement set for this proof of concept was just cut to the minimum bits to proof it working, with a lot of room for future improvements. Those changes in the scope of it, let us finish the work, though, so they ended up being a critical choice to make.

6.4. Future work

As said during the work, there have been many areas where we had not enough time to either develop since the beginning, or we had to drop out for time reasons, in either case there are many topics to be developed:

- User authentication and authorization: our solution even if it works, is far from convenient if we think about it being used by end users, this is why we would like to explore this further and find some solution or mechanism that allows for a much friendly usage while keeping the security and anonymity at its maximum.
- Data storage quotas: in any realistic scenario, a user would not want to fill its disk with other users data. Would be interesting to find a way to let users make quotas on disk usage. To avoid abuse (people storing a lot of files while keeping few for others) some rule or karma based system needs to be in place to incentivize people to lend storage if they are using the system.
- More operations: right now the system allows only for adding files, would be nice for a more fine grained set of operations, at a minimum, deleting is a requirement.
- Live file sync: would be interesting to explore anything required to let users define where the files are stored, and sync automatically anything put or removed from this folder, also mounting volumes and maybe integrating natively with FUSE without the read only constraints IPFS has.
- IP self discovery and NAT by passing: currently even if peer discovery works over IRC and is able to bypass NAT setups, the address shared is local. This mean we need some means to find our public IP and share this one instead, and also be sure to bypass NAT. IPFS has NAT bypassing capabilities, but this is different with the IPFS Cluster bit, which would need to be developed.
- Improved peer discovery: find better ways for peer discovery that scales, and as far as possible it is decentralized.

8. Glossary

- *InterPlanetary FileSystem:* The InterPlanetary FileSystem (IPFS) is a protocol and peer-to-peer network for storing and sharing data in a distributed file system. [2]
- *IPFS:* InterPlanetary FileSystem
- *Juan Benet*: author of IPFS paper and founder CEO of Protocol Labs.
- *Open Source Software*: Open source software is software with source code that anyone can inspect, modify, and enhance. [4]
- *OSS*: Open Source Software
- *Overlay network*: network built on top of another network.
- *Protocol Labs*: a research and development company in charge of the development of IPFS and other technologies.
- *Peer discovery*: process that P2P networks use for finding other nodes.
- *Bootstrap servers*: are servers usually provided in advance, which purpose is to provide new nodes with a list of peers to communicate with.
- *AES256:* symmetric encryption algorithm developed by the USA government.
- *Bitcoin:* a virtual currency based on a distributed and decentralized network.
- *VPN:* a virtual private network, let computers communicate as if they were in a local network over the internet.
- *NAT*: network address translation is a mechanism used to communicate between networks with incompatible IP ranges, for example a local network and the Internet.
- *IRC:* internet relay chat, an old protocol used to have chat rooms where people talked with others.
- *Freenode*: an IRC network focused mainly in open source development.
- *ISO week:* the number of the week in the current year according to the standard ISO 8601.
- *Go package*: in the Go language, a package is a set of files that hold code together, the code inside usually have a shared semantic meaning, i.e.: math package, encoding package, …
- *Symmetric key encryption:* in this encryption method, the same key is used for encrypting and decrypting.

9. Bibliography

[1] Juan Benet.
IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3), 2014
https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf
[2] https://en.wikipedia.org/wiki/InterPlanetary_File_System, edited 13 Oct 2019
[3] https://github.com/ipfs/ipfs/blob/master/REQUIREMENTS.md, edited 6 Nov 2018
[4] https://opensource.com/resources/what-open-source, visited 15 Oct 2019
[5] https://en.wikipedia.org/wiki/Distributed_hash_table, edited 29 Oct 2019
[6] https://tlu.tarilabs.com/protocols/dht/MainReport.html, visited 15 Oct 2019
[7] https://en.wikipedia.org/wiki/Merkle_tree, edited 2 Nov 2019
[8] https://ldapwiki.com/wiki/Merkle%20DAG, edited 17 Sep 2019
[9] https://medium.com/textileio/how-the-ipfs-dht-works-47af8bfd3c6a, edited 18 Jul 2019
[10] David Mazières
Self-Certifying FileSystem, 2000
https://www.scs.stanford.edu/home/papers/mazieres:thesis.ps.gz
[11] https://golden.com/wiki/Directed_acyclic_graph, visited 15 Oct 2019
[12] https://es.wikipedia.org/wiki/BitTorrent, visited 15 Oct 2019
[13] Baumgart, Ingmar & Mies, Sergio. (2008). S/Kademlia: A practicable approach towards secure key-based routing. 2. 1-8. 10.1109/ICPADS.2007.4447808.
[14] Freedman M.J., Maziéres D. (2003) Sloppy Hashing and Self-Organizing Clusters. In: Kaashoek M.F., Stoica I. (eds) Peer-to-Peer Systems II. IPTPS 2003. Lecture Notes in Computer Science, vol 2735. Springer, Berlin, Heidelberg
[15] https://cluster.ipfs.io/, visited 10 Oct 2019
[16] https://en.wikipedia.org/wiki/Satoshi_Nakamoto, visited 05 Jan 2020
[17] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard, visited 05 Jan 2020
[18] https://en.wikipedia.org/wiki/Multicast_DNS, visited 04 Jan 2020
[19] https://github.com/bitcoin/bitcoin/commit/c2efd981aa14e94cce4a0a888b6ee1f4e4347924, visited 04 Jan 2020
[20] https://freenode.net/, visited 23 Dec 2019
[21] https://www.iso.org/home.html, visited 23 Dec 2019
[22] https://ed25519.cr.yp.to/, visited 23 Dec 2019
[23] https://docs.ipfs.io/reference/api/cli/#ipfs-key-gen, visited 23 Dec 2019
[24] https://golang.org/, visited 23 Dec 2019
[25] https://www.docker.com/, visited 02 Jan 2020
[26] https://www.redhat.com/es/topics/containers/whats-a-linux-container, visited on 02 Jan 2020
[27] https://en.wikipedia.org/wiki/Linux_namespaces, updated 19 Dec 2019
[28] https://www.electronicdesign.com/technologies/dev-tools/article/21801722/whats-the-difference-between-containers-and-virtual-machines, published 15 Jul 2016
[29] https://docs.docker.com/compose/, visited 02 Jan 2020
[30] https://tahoe-lafs.org/trac/tahoe-lafs, visited 02 Jan 2020
[31] https://www.rubrik.com/blog/erasure-coding-rubrik-doubled-capacity-cluster/, published 28 Feb 2017
[32] https://tahoe-lafs.readthedocs.io/en/tahoe-lafs-1.12.1/architecture.html#capabilities, visited 02 Jan 2020
[33] https://freenetproject.org/, visited 02 Jan 2020
[34] https://en.wikipedia.org/wiki/Freenet#Technical_design, visited 02 Jan 2020